CHAPTER 9
Linear Algebra, Nonparametric Statistics, and Time Series Analysis

Introduction

This chapter explores the essential mathematical foundations, statistical techniques, and methods for analyzing time-dependent data. We will cover three interconnected topics: linear algebra, nonparametric statistics, and time series analysis, incorporating survival analysis. The journey begins with linear algebra, where we will unravel key concepts such as linear functions, vectors, and matrices, providing a solid framework for understanding complex data structures. Nonparametric statistics will enable us to analyze data without the restrictive assumptions of parametric models. We will explore techniques like rank-based tests and kernel density estimation, which offer flexibility in analyzing a wide range of data types.

Time series data, prevalent in diverse areas such as stock prices, weather patterns, and heart rate variability, will be examined with a focus on trend and seasonality analysis. In

the realm of survival analysis, where life events such as disease progression, customer churn, or equipment failure are unpredictable, we will delve into the analysis of time-to-event data. We will demystify techniques such as Kaplan-Meier estimators, making survival analysis accessible and understandable. Throughout the chapter, each concept will be illustrated with practical examples and real-world applications, providing a hands-on guide for implementation.

Structure

In this chapter, we will discuss the following topics:

- Linear algebra
- Nonparametric statistics
- Survival analysis
- Time series analysis

Objectives

This chapter provides the reader with the necessary tools, the ability to gain insight, the understanding of the theory and the ways to implement linear algebra, nonparametric statistics and time series analysis techniques with Python. By the last page, you will be armed with the knowledge to tackle complex data challenges and interpret results with clarity about these topics.

Linear algebra

Linear algebra is a branch of mathematics that focuses on the study of vectors, vector spaces and linear transformations. It deals with linear equations, linear functions and their representations through matrices and determinants.

Let us understand vectors, linear function and matrices in

linear algebra.

Following is the explanation of vectors:

- **Vectors:** Vectors are a fundamental concept in linear algebra as they represent quantities that have both magnitude and direction. Examples of such quantities include velocity, force and displacement. In statistics, vectors organize data points. Each data point can be represented as a vector, where each component corresponds to a specific feature or variable.

Tutorial 9.1: To create a 2D vector with NumPy and display, is as follows:

```
import numpy as np
# Create a 2D vector
v = np.array([3, 4])
# Access individual components
x, y = v
# Calculate magnitude (Euclidean norm) of the vecto
r
magnitude = np.linalg.norm(v)
print(f"Vector v: {v}")
print(f"Components: x = {x}, y = {y}")
print(f"Magnitude: {magnitude:.2f}")
```

Output:

```
Vector v: [3 4]
Components: x = 3, y = 4
Magnitude: 5.00
```

- **Linear function**: A linear function is represented by the equation $f(x) = ax + b$, where a and b are constants. They model relationships between variables. For example, linear regression shows how a dependent variable changes linearly with respect to an independent variable.

Tutorial 9.2: To create a simple linear function, $f(x) =$

2x + 3 and plot it, is as follows:

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3. # Define a linear function: f(x) = 2x + 3
4. def linear_function(x):
5.     return 2 * x + 3
6. # Generate x values
7. x_values = np.linspace(-5, 5, 100)
8. # Calculate corresponding y values
9. y_values = linear_function(x_values)
10. # Plot the linear function
11. plt.plot(x_values, y_values, label="f(x) = 2x + 3")
12. plt.xlabel("x")
13. plt.ylabel("f(x)")
14. plt.title("Linear Function")
15. plt.grid(True)
16. plt.legend()
17. plt.savefig("linearfunction.jpg",dpi=600,bbox_inches
    ='tight')
18. plt.show()
```

Output:

It plots the *f(x) = 2x + 3* as shown in *Figure 9.1:*

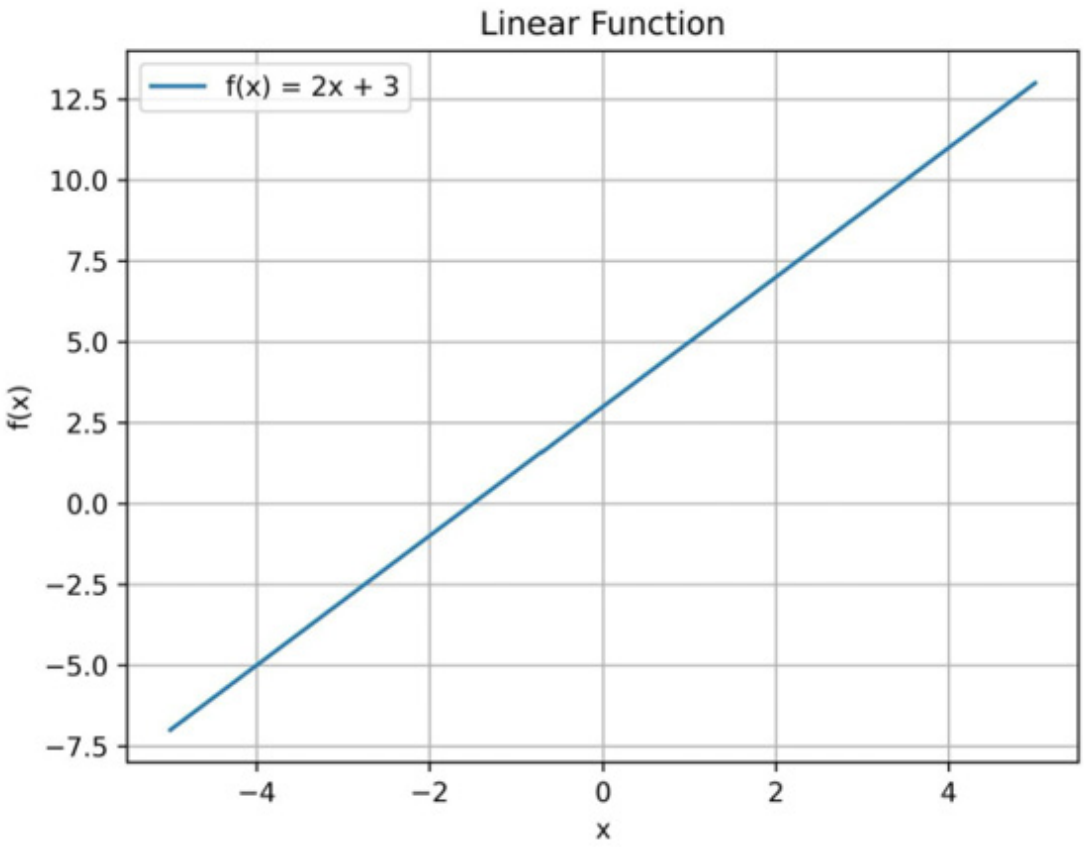

Figure 9.1: *Plot of a linear function*

- **Matrices**: Matrices are rectangular arrays of numbers that are commonly used to represent systems of linear equations and transformations. In statistics, matrices are used to organize data, where rows correspond to observations and columns represent variables. For example, a dataset with height, weight, and age can be represented as a matrix.

  Tutorial 9.3: To create a matrix (rectangular array) of numbers with NumPy and transpose it, as follows:

```
import numpy as np
# Create a 2x3 matrix
A = np.array([[1, 2, 3],
              [4, 5, 6]])
# Access individual elements
element_23 = A[1, 2]
```

```
# Transpose the matrix
A_transposed = A.T
print(f"Matrix A:\n{A}")
print(f"Element at row 2, column 3: {element_23}")
print(f"Transposed matrix A:\n{A_transposed}")
```

Output:

```
Matrix A:
[[1 2 3]
 [4 5 6]]
Element at row 2, column 3: 6
Transposed matrix A:
[[1 4]
 [2 5]
 [3 6]]
```

Linear algebra models and analyses relationships between variables, aiding our comprehension of how changes in one variable affect another. Its further application include cryptography to create solid encryption techniques, regression analysis, dimensionality reduction and solving systems of linear equations. We discussed this earlier in *Chapter 7, Statistical Machine Learning* on linear regression. For example, imagine we want to predict a person's weight based on their height. We collect data from several individuals and record their heights (in inches) and weights (in pounds). Linear regression allows us to create a straight line (a linear model) that best fits the data points (height and weight). Using this method, we can predict someone's weight based on their height using the linear equation. The use and implementation of linear algebra in statistics is shown in the following tutorials:

Tutorial 9.4: To illustrate the use of linear algebra, solve a linear system of equations using the linear algebra submodule of SciPy, is as follows:

```
import numpy as np
# Import the linear algebra submodule of SciPy and assig
n it the alias "la"
import scipy.linalg as la
A = np.array([[1, 2], [3, 4]])
b = np.array([3, 17])
# Solving a linear system of equations
x = la.solve(A, b)
print(f"Solution x: {x}")
print(f"Check if A @ x equals b: {np.allclose(A @ x, b)}")
```

Output:

```
Solution x: [11. -4.]
Check if A @ x equals b: True
```

Tutorial 9.5: To illustrate the use of linear algebra in statistics to compare performance, solving vs. inverting for linear systems, using SciPy, is as follows:

```
import numpy as np
import scipy.linalg as la
A1 = np.random.random((1000, 1000))
b1 = np.random.random(1000)
# Uses %timeit magic command to measure the executio
n time of la.solve(A1, b1) and la.solve solves linear equat
ions
solve_time = %timeit -o la.solve(A1, b1)
# Measures the time for solving by first inverting A1 usi
ng la.inv(A1) and then multiplying the inverse with b1.
inv_time = %timeit -o la.inv(A1) @ b1
# Prints the best execution time for la.solve method in m
illiseconds
print(f"Solve time: {solve_time.best:.2f} ms")
# Prints the best execution time for the inversion metho
d in milliseconds
```

```
print(f"Inversion time: {inv_time.best:.2f} ms")
```

Output:

```
31.3 ms ± 4.05 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
112 ms ± 4.51 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
Solve time: 0.03 ms
Inversion time: 0.11 ms
```

Tutorial 9.6: To illustrate the use of linear algebra in statistics to perform basic matrix properties, using the linear algebra submodule of SciPy, is as follows:

```
import numpy as np
import scipy.linalg as la
# Create a complex matrix C
C = np.array([[1, 2 + 3j], [3 - 2j, 4]])
# Print the conjugate of C (element-wise complex conjugate)
print(f"Conjugate of C:\n{C.conjugate()}")
# Print the trace of C (sum of diagonal elements)
print(f"Trace of C: {np.diag(C).sum()}")
# Print the matrix rank of C (number of linearly independent rows/columns)
print(f"Matrix rank of C: {np.linalg.matrix_rank(C)}")
# Print the Frobenius norm of C (square root of sum of squared elements)
print(f"Frobenius norm of C: {la.norm(C, None)}")
# Print the largest singular value of C (largest eigenvalue of C*C.conjugate())
print(f"Largest singular value of C: {la.norm(C, 2)}")
# Print the smallest singular value of C (smallest eigenvalue of C*C.conjugate())
print(f"Smallest singular value of C: {la.norm(C, -2)}")
```

Output:

```
Conjugate of C:
[[1.-0.j 2.-3.j]
 [3.+2.j 4.-0.j]]
Trace of C: (5+0j)
Matrix rank of C: 2
Frobenius norm of C: 6.557438524302
Largest singular value of C: 6.389028023601217
Smallest singular value of C: 1.476590977094992
```

Tutorial 9.7: To illustrate the use of linear algebra in statistics to compute the least squares solution in a square matrix, using the linear algebra submodule of SciPy, is as follows:

```
import numpy as np
import scipy.linalg as la
# Define a square matrix A1 and vector b1
A1 = np.array([[1, 2], [2, 4]])
b1 = np.array([3, 17])
# Attempt to solve the system of equations A1x = b1 using la.solve
try:
    x = la.solve(A1, b1)
    print(f"Solution using la.solve: {x}")  # Print solution if successful
except la.LinAlgError as e:  # Catch potential error if matrix is singular
    print(f"Error using la.solve: {e}")  # Print error message
# # Compute least-squares solution
x, residuals, rank, s = la.lstsq(A1, b1)
print(f"Least-squares solution x: {x}")
```

Output:

```
Error using la.solve: Matrix is singular.
Least-squares solution x: [1.48 2.96]
```

Tutorial 9.8: To illustrate the use of linear algebra in statistics to compute the least squares solution of a random matrix, using the linear algebra submodule of SciPy, is as follows:

```
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt
A2 = np.random.random((10, 3))
b2 = np.random.random(10)
#Computing least square from random matrix
x, residuals, rank, s = la.lstsq(A2, b2)
print(f"Least-squares solution for random A2: {x}")
```

Output:

```
Least-
squares solution for random A2: [0.34430232 0.5421179
6 0.18343947]
```

Tutorial 9.9: To illustrate the implementation of linear regression to predict car prices based on historical data, is as follows:

```
import numpy as np
from scipy import linalg
# Sample data: car prices (in thousands of dollars) and f
eatures
prices = np.array([20, 25, 30, 35, 40])
features = np.array([[2000, 150],
                     [2500, 180],
                     [2800, 200],
                     [3200, 220],
                     [3500, 240]])
# Fit a linear regression model
```

```
coefficients, residuals, rank, singular_values = linalg.lstsq(features, prices)
# Predict price for a new car with features [3000, 170]
new_features = np.array([3000, 170])
# Calculate predicted price using the dot product of the new features and their corresponding coefficients
predicted_price = np.dot(new_features, coefficients)
print(f"Predicted price: ${predicted_price:.2f}k")
```

Output:

```
Predicted price: $41.60k
```

Nonparametric statistics

Nonparametric statistics is a branch of statistics that does not rely on specific assumptions about the underlying probability distribution. Unlike parametric statistics, which assume that data follow a particular distribution (such as the normal distribution), nonparametric methods are more flexible and work well with different types of data. Nonparametric statistics make inferences without assuming a particular distribution. They often use ordinal data (based on rankings) rather than numerical values. As mentioned unlike parametric methods, nonparametric statistics do not estimate specific parameters (such as mean or variance) but focus on the overall distribution.

Let us understand nonparametric statistics and its use through an example of clinical trial rating, as follows:

- **Clinical trial rating:** Imagine that a researcher is conducting a clinical trial to evaluate the effectiveness of a new pain medication. Participants are asked to rate their treatment experience on a scale of one to five (where one is *very poor* and five is *excellent*). The data collected consist of ordinal ratings, not continuous numerical values. These ratings are inherently

nonparametric because they do not follow a specific distribution.

- To analyze the treatment's impact, the researcher can apply nonparametric statistical tests like the Wilcoxon signed-rank test. Wilcoxon signed-rank test is a statistical method used to compare paired data, specifically when you want to assess whether there is a significant difference between two related groups. It compares the median ratings before and after treatment and does not assume a normal distribution and is suitable for paired data.
- **Hypotheses**:
    - **Null hypothesis (H_0)**: The median rating before treatment is equal to the median rating after treatment.
    - **Alternative hypothesis (H_1)**: The median rating differs before and after treatment.
- If the p-value from the test is small (typically less than 0.05), we reject the null hypothesis, indicating a significant difference in treatment experience.

This example shows that nonparametric methods allow us to make valid statistical inferences without relying on specific distributional assumptions. They are particularly useful when dealing with ordinal data or situations where parametric assumptions may not hold.

Tutorial 9.10: To illustrate the use of nonparametric statistics to compare treatment ratings (ordinal data). We collect treatment ratings (ordinal data) before and after a new drug. We want to know if the drug improves the patient's experience, as follows:

```
1. import numpy as np
2. from scipy.stats import wilcoxon
3. # Example data (ratings on a scale of 1 to 5)
```

```
before_treatment = [3, 4, 2, 3, 4]
after_treatment = [4, 5, 3, 4, 5]
# Null Hypothesis (H₀): The median treatment rating before the new drug is equal to the median rating after the drug.
# Alternative Hypothesis (H₁): The median rating differs before and after the drug.
# Perform Wilcoxon signed-rank test
statistic, p_value = wilcoxon(before_treatment, after_treatment)
if p_value < 0.05:
    print("P-value:", p_value)
    print("P-value is less than 0.05, so reject the null hypothesis, we can confidently say that the new drug led to better treatment experience.")
else:
    print("P-value:", p_value)
    print("No significant change")
    print("P value is greater than or equal to 0.05, so we cannot reject the null hypothesis and therefore cannot conclude that the drug had a significant effect.")
```

Output:

```
P-value: 0.0625
No significant change
P value is greater than or equal to 0.05, so we cannot reject the null hypothesis and therefore cannot conclude

that the drug had a significant effect.
```

Nonparametric statistics relies on statistical methods that do not assume a specific distribution for the data, making them versatile for a wide range of applications where traditional parametric assumptions may not hold. In this

section, we will explore some key nonparametric methods, including **rank-based tests**, **goodness-of-fit tests**, and **independence tests**. Rank-based tests, such as the **Kruskal-Wallis test**, allow for comparisons across groups without relying on parametric distributions. **Goodness-of-fit tests**, like the chi-square test, assess how well observed data align with expected distributions, while **independence tests**, such as Spearman's rank correlation or Fisher's exact test, evaluate relationships between variables without assuming linearity or normality. Additionally, resampling techniques like **bootstrapping** provide robust estimates of confidence intervals and other statistics, bypassing the need for parametric assumptions. These nonparametric methods are essential tools for data analysis when distributional assumptions are difficult to justify. Let us explore some key nonparametric methods:

Rank-based tests

Rank-based tests compare rankings or orders of data points between groups. It includes Mann-Whitney U test (Wilcoxon rank-sum test) and Wilcoxon signed-rank test. The Mann-Whitney U test compares medians between two independent groups (e.g., treatment vs. control group). It determines if their distributions differ significantly and is useful when assumptions of normality are violated. Wilcoxon signed-rank test compares paired samples (e.g., before and after treatment), as in *Tutorial 9.10*. It tests if the median difference is zero and is robust to non-gaussian data.

Goodness-of-fit tests

Goodness-of-fit tests assess whether observed data fits a specific distribution. It includes chi-squared goodness-of-fit test. This test checks if observed frequencies match expected frequencies in different categories. Suppose you are a data analyst working for a shop owner who claims that

an equal number of customers visit the shop each weekday. To test this hypothesis, you record the number of customers that come into the shop during a given week, as follows:

Days	Monday	Tuesday	Wednesday	Thursday	Friday
Number of Customers	50	60	40	47	53

***Table 9.1**: Number of customers per week days*

Using this data, we determine whether the observed distribution of customers across weekdays matches the expected distribution (equal number of customers each day).

Tutorial 9.11: To implement chi-square goodness of fit test to see if the observed distribution of customers across weekdays matches the expected distribution (equal number of customers each day), is as follows:

```
1. import scipy.stats as stats
2. # Create two arrays to hold the observed and expected number of customers for each day
3. expected = [50, 50, 50, 50, 50]
4. observed = [50, 60, 40, 47, 53]
5. # Perform Chi-
   Square Goodness of Fit Test using chisquare function
6. # Null Hypothesis (H₀): The variable follows the hypothesized distribution (equal number of customers each day).
7. # Alternative Hypothesis (H₁): The variable does not follow the hypothesized distribution.
8. # Chisquare function takes two arrays: f_obs (observed counts) and f_exp (expected counts).
9. # By default, it assumes that each category is equally likely.
10. result = stats.chisquare(f_obs=observed, f_exp=expected)
```

```
print("Chi-Square Statistic:", round(result.statistic, 3))
print("p-value:", round(result.pvalue, 3))
```

Output:

```
Chi-Square Statistic: 4.36
p-value: 0.359
```

The chi-square test statistic is calculated as 4.36, and the corresponding p-value is 0.359. Since the p-value is not less than 0.05 (our significance level), we fail to reject the null hypothesis. This means we do not have sufficient evidence to say that the true distribution of customers is different from the distribution claimed by the shop owner.

Independence tests

Independence tests determine if two categorical variables are independent. It includes chi-squared test of independence and Kendall's tau or Spearman's rank correlation. Chi-squared test of independence examines association between variables in a contingency table, as discussed in earlier in *Chapter 6*, *Hypothesis Testing and Significance Tests*. Kendall's tau or Spearman's rank correlation assess correlation between ranked variables.

Suppose two basketball coaches rank 12 players from worst to best. The rankings assigned by each coach are as follows:

Players	Coach #1 Rank	Coach #2 Rank
A	1	2
B	2	1
C	3	3
D	4	5
E	5	4
F	6	6

G	7	8
H	8	7
I	9	9
J	10	11
K	11	10
L	12	12

***Table 9.2**: Rankings assigned by each coach*

Using this we can calculate Kendall's Tau, let us calculate Kendall's Tau to assess the correlation between the two coaches' rankings. A positive Tau indicates a positive association, while a negative tau indicates a negative association. The closer Tau is to 1 or -1, the stronger the association. A Tau of zero indicates no association.

Tutorial 9.12: To calculate Kendall's Tau to assess the correlation between the two coaches' rankings, is as follows:

```
1. import scipy.stats as stats
2. # Coach #1 rankings
3. coach1_ranks = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
4. # Coach #2 rankings
5. coach2_ranks = [2, 1, 3, 5, 4, 6, 8, 7, 9, 11, 10, 12]
6. # Calculate concordant and discordant pairs
7. concordant = 0
8. discordant = 0
9. n = len(coach1_ranks)
10. # Iterate through all pairs of players (i, j) where i < j
11. for i in range(n):
12.     for j in range(i + 1, n):
13.         # Check if both coaches ranked player i higher tha
    n player j (concordant pair)
```

```
        # or both coaches ranked player i lower than player j (also concordant pair)
        if (coach1_ranks[i] < coach1_ranks[j] and coach2_ranks[i] < coach2_ranks[j]) or \
           (coach1_ranks[i] > coach1_ranks[j] and coach2_ranks[i] > coach2_ranks[j]):
            concordant += 1
        # Otherwise, it's a discordant pair
        elif (coach1_ranks[i] < coach1_ranks[j] and coach2_ranks[i] > coach2_ranks[j]) or \
             (coach1_ranks[i] > coach1_ranks[j] and coach2_ranks[i] < coach2_ranks[j]):
            discordant += 1
# Calculate Kendall's Tau
tau = (concordant - discordant) / (concordant + discordant)
print("Kendall's Tau:", round(tau, 3))
```

Output:

```
Kendall's Tau: 0.879
```

Kendall's Tau of 0.879 indicates a strong positive association between the two ranked variables. In other words, the rankings assigned by the two coaches are closely related, and their preferences align significantly.

Kruskal-Wallis test

Kruskal-Wallis test is nonparametric alternative to one-way ANOVA. It allows to compare medians across multiple independent groups and generalizes the Mann-Whitney test. Suppose researchers want to determine if three different fertilizers lead to different levels of plant growth. They randomly select 30 different plants and split them into three groups of 10, applying a different fertilizer to each group. After one month, they measure the height of each plant.

Tutorial 9.13: To implement the Kruskal-Wallis test to compare median heights across multiple groups, is as follows:

```
from scipy import stats
# Create three arrays to hold the plant measurements for each of the three groups
group1 = [7, 14, 14, 13, 12, 9, 6, 14, 12, 8]
group2 = [15, 17, 13, 15, 15, 13, 9, 12, 10, 8]
group3 = [6, 8, 8, 9, 5, 14, 13, 8, 10, 9]
# Perform Kruskal-Wallis Test
# Null hypothesis (H₀): The median is equal across all groups.
# Alternative hypothesis (Hₐ): The median is not equal across all groups
result = stats.kruskal(group1, group2, group3)
print("Kruskal-Wallis Test Statistic:", round(result.statistic, 3))
print("p-value:", round(result.pvalue, 3))
```

Output:

```
Kruskal-Wallis Test Statistic: 6.288
p-value: 0.043
```

Here, p-value is less than our chosen significance level (e.g., 0.05), so we reject the null hypothesis. We conclude that the type of fertilizer used leads to statistically significant differences in plant growth.

Bootstrapping

Bootstrapping is a resampling technique to estimate parameters or confidence intervals. Like bootstrapping the mean or median from a sample. Bootstrapping is a resampling technique that generates simulated samples by repeatedly drawing from the original dataset. Each simulated sample is the same size as the original sample. By

creating these simulated samples, we can explore the variability of sample statistics and make inferences about the population. It is especially useful when population distribution is unknown or does not follow a standard form. Sample sizes are small. You want to estimate parameters (e.g., mean, median) or construct confidence intervals.

For example, imagine we have a dataset of exam scores (sampled from an unknown population). We resample the exam scores with replacement to create bootstrap samples. We want to estimate the mean exam score and create a bootstrapped confidence interval. The bootstrapped mean provides an estimate of the population mean. The confidence interval captures the uncertainty around this estimate.

Tutorial 9.14: To implement nonparametric statistical method bootstrapping to bootstrap the mean or median from a sample, is as follows:

```
import numpy as np
# Example dataset (exam scores)
scores = np.array([78, 85, 92, 88, 95, 80, 91, 84, 89, 87])
# Number of bootstrap iterations
# The bootstrapping process is repeated 10,000 times (10,000 iterations is somewhat arbitrary).
# Allowing us to explore the variability of the statistic (mean in this case). And construct confidence intervals.
n_iterations = 10_000
# Initialize an array to store bootstrapped means
bootstrapped_means = np.empty(n_iterations)
# Perform bootstrapping
for i in range(n_iterations):
    bootstrap_sample = np.random.choice(scores, size=len(scores), replace=True)
```

```
    bootstrapped_means[i] = np.mean(bootstrap_sample)
# Calculate the bootstrap means of all bootstrapped samples from the main exam score data set
print(f"Bootstrapped Mean: {np.mean(bootstrapped_means):.2f}")
# Calculate the 95% confidence interval
lower_bound = np.percentile(bootstrapped_means, 2.5)
upper_bound = np.percentile(bootstrapped_means, 97.5)
print(f"95% Confidence Interval: [{lower_bound:.2f}, {upper_bound:.2f}]")
```

Output:

```
Bootstrapped Mean: 86.89
95% Confidence Interval: [83.80, 90.00]
```

This means that we expect the average exam score in the entire population (from which our sample was drawn) to be around 86.89. We are 95% confident that the true population mean exam score falls within this interval (83.80, 89.90).

The nonparametric methods include **Kernel Density Estimation (KDE)** which is nonparametric way to estimate probability density functions (probability distribution for a random, continuous variable) and is useful for visualizing data distributions. The survival analysis is also a nonparametric method because it focuses on estimating survival probabilities without making strong assumptions about the underlying distribution of event times. Kaplan-Meier estimator is a non-parametric method used to estimate the survival function.

Survival analysis

Survival analysis is a statistical method used to analyze the amount of time it takes for an event of interest to occur (helping to understand the time it takes for an event to

occur). It is also known as time-to-event analysis or duration analysis. Common applications include studying time to death (in medical research), disease recurrence, or other significant events. But not limited to medicine, it can be used in various fields such as finance, engineering and social sciences. For example, imagine a clinical trial for lung cancer patients. Researchers want to study the time until death (survival time) for patients receiving different treatments. Other examples include analyzing time until finding a new job after unemployment, mechanical system failure, bankruptcy of a company, pregnancy & recovery from a disease.

Kaplan-Meier estimator is one of the most widely used and simplest methods of survival analysis. It handles censored data, where some observations are partially observed (e.g., lost to follow-up). Kaplan-Meier estimation includes the following:

- Sort the data by time
- Calculate the proportion of surviving patients at each time point
- Multiply the proportions to get the cumulative survival probability
- Plot the survival curve

For example, imagine that video game players are competing in a battle video game tournament. The goal is to use survival analysis to see which player can stay **alive** (not killed) the longest.

In the context of survival analysis, data censoring is often encountered concept. Sometimes we do not observe the event for the entire study period, which is when censoring comes into play. Censored data is now; the organizer may have to end the game early. In this case, some player may still be alive when the game end whistle blows. We know they survived at least that long, but we do not know exactly how much longer they would have lasted. This is censored

data in survival analysis. Censoring has type fight and left. Right-censored data occurs when we know an event has not happened yet, but we do not know exactly when it will happen in the future. Here censored data can have type right-centered and left centered like in above video game competition. Players who were alive in the game when the whistle blew are right-censored. We know that they survived at least that long (until the whistle blew), but their true survival time (how long they would have survived if the game had continued) is unknown. Left censored data is the opposite of right-censored data. It occurs when we know that an event has already happened, but we do not know exactly when it happened in the past.

Tutorial 9.15: To implement the Kaplan-Meier method to estimate the survival function (survival analysis) of a video game player in a battling video game competition, is as follows:

```
from lifelines import KaplanMeierFitter
import numpy as np
import matplotlib.pyplot as plt
# Let's create a sample dataset
# durations represents the time of the event (e.g., time until student is "alive" in game (not tagged)
# event_observed is a boolean array that denotes if the event was observed (True) or censored (False)
durations = [24, 18, 30, 12, 36, 15, 8, 42, 21, 6,
             10, 27, 33, 5, 19, 45, 28, 9, 39, 14,
             22, 7, 48, 31, 17, 20, 40, 25, 3, 37]
event_observed = [0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0,
                  1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1]
# Create an instance of KaplanMeierFitter
kmf = KaplanMeierFitter()
# Fit the data into the model
```

```
kmf.fit(durations, event_observed)
# Plot the survival function
kmf.plot_survival_function()
# Customize plot (optional)
plt.xlabel('Time')
plt.ylabel('Survival Probability')
plt.title('Kaplan-Meier Survival Curve')
plt.grid(True)
# Save the plot
plt.savefig('kaplan_meier_survival.png', dpi=600, bbox_inches='tight')
plt.show()
```

Output:

Figure 9.2 and *Figure 9.3* show the probability of survival appears to decrease over time with a steeper decline observed in the time period near 10 to near 40 points. This suggests that patients are more likely to experience the event (possibly death) as time progresses after surgery. The KM_estimate in *Figure 9.2* is survival curve line, this line represents the Kaplan-Meier survival curve, which is estimated survival probability over time. And shaded area is the **Confidence Interval (CI).** The narrower the CI, the more precise our estimate of the survival curve. If the CI widens at certain points, it indicates greater uncertainty in the survival estimate at those time intervals.

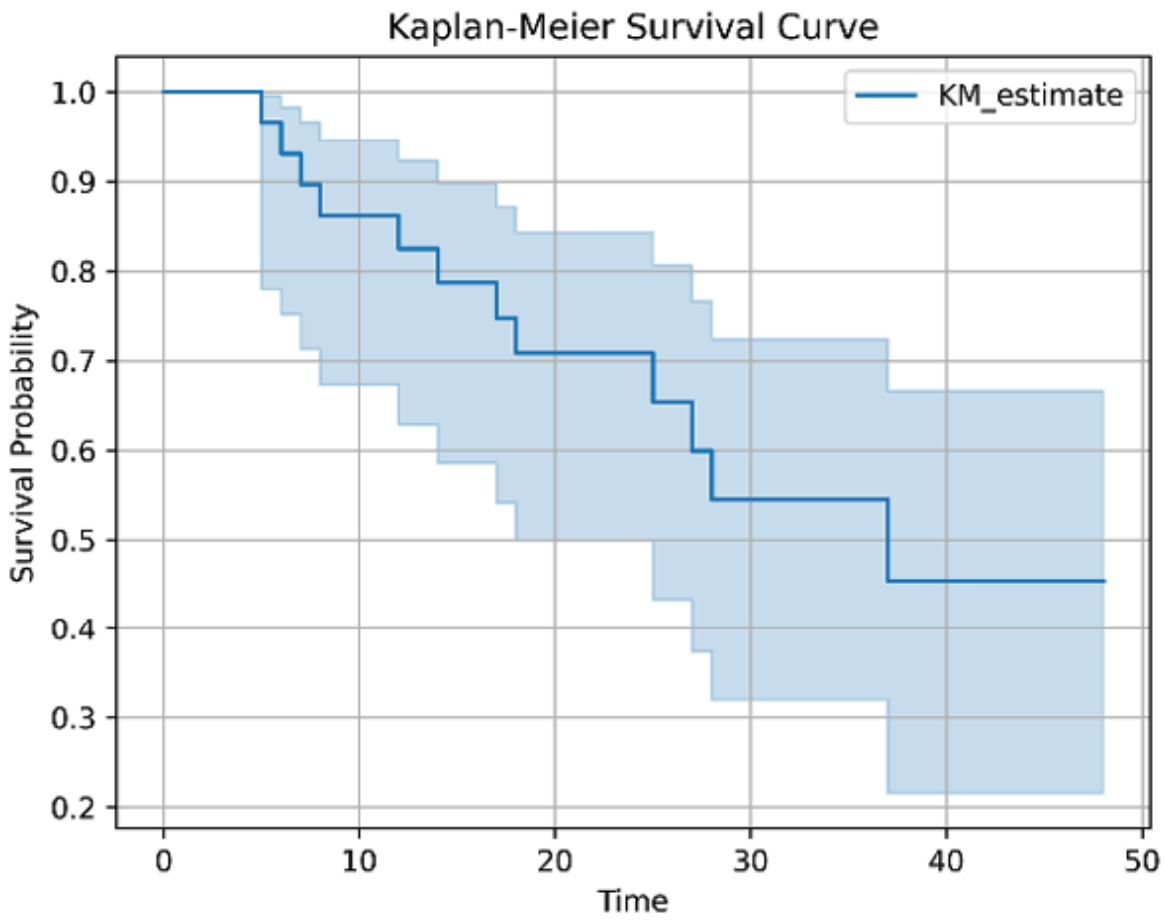

Figure 9.2: *Kaplan-Meier curve showing change in probability of survival over time*

Let us see another example, suppose we want to estimate the lifespan of patients (time until death) with certain conditions using a sample dataset of 30 patients with their IDs, time of observation (in months) and event status (alive or death). Let us say we are studying patients with heart failure. We will follow them for two years to see if they have a heart attack during that time.

Following is our data set:

- **Patient A:** Has a heart attack after six months (event observed).
- **Patient B**: Still alive after two years (right censored).
- **Patient C**: Drops out of the study after one year (right

censored).

In this case, the way censoring works is as follows:

- **Patient A**: We know the exact time of the event (heart attack).
- **Patient B**: Their data are right-censored because we did not observe the event (heart attack) during the study.
- **Patient C**: Also, right-censored because he dropped out before the end of the study.

Tutorial 9.16: To implement Kaplan-Meier method to estimate survival function (survival analysis) of the patients with a certain condition over time, is as follows:

```
import matplotlib.pyplot as plt
import pandas as pd
# Import Kaplan Meier Fitter from the lifelines library
from lifelines import KaplanMeierFitter
# Create sample healthcare data (change names as needed)
data = pd.DataFrame({
    # IDs from 1 to 10
    "PatientID": range(1, 31),
    # Time is how long a patient was followed up from the start of the study,
    # until the end of the study or the occurrence of the event.
    "Time": [24, 18, 30, 12, 36, 15, 8, 42, 21, 6,
             10, 27, 33, 5, 19, 45, 28, 9, 39, 14,
             22, 7, 48, 31, 17, 20, 40, 25, 3, 37],
    # Event indicates the event status of patient at the end of observation ,
    # weather patient was dead or alive at the end of study period
    "Event": ['Alive', 'Death', 'Alive', 'Death', 'Alive', 'Alive'
```

```
, 'Death', 'Alive', 'Alive', 'Death',
                'Alive', 'Death', 'Alive', 'Death', 'Alive', 'Alive', 'Death', 'Alive', 'Alive', 'Death',
                'Alive', 'Death', 'Alive', 'Alive', 'Death', 'Alive', 'Alive', 'Death', 'Alive', 'Death']
})
# Convert Event to boolean (Event indicates occurrence of death)
data["Event"] = data["Event"] == "Death"
# Create Kaplan-Meier object (focus on event occurrence)
kmf = KaplanMeierFitter()
kmf.fit(data["Time"], event_observed=data["Event"])
# Estimate the survival probability at different points
time_points = range(0, max(data["Time"]) + 1)
survival_probability = kmf.survival_function_at_times(time_points).values
# Plot the Kaplan-Meier curve
plt.step(time_points, survival_probability, where='post')
plt.xlabel('Time (months)')
plt.ylabel('Survival Probability')
plt.title('Kaplan-Meier Curve for Patient Survival')
plt.grid(True)
plt.savefig('Survival_Analysis2.png', dpi=600, bbox_inches='tight')
plt.show()
```

Output:

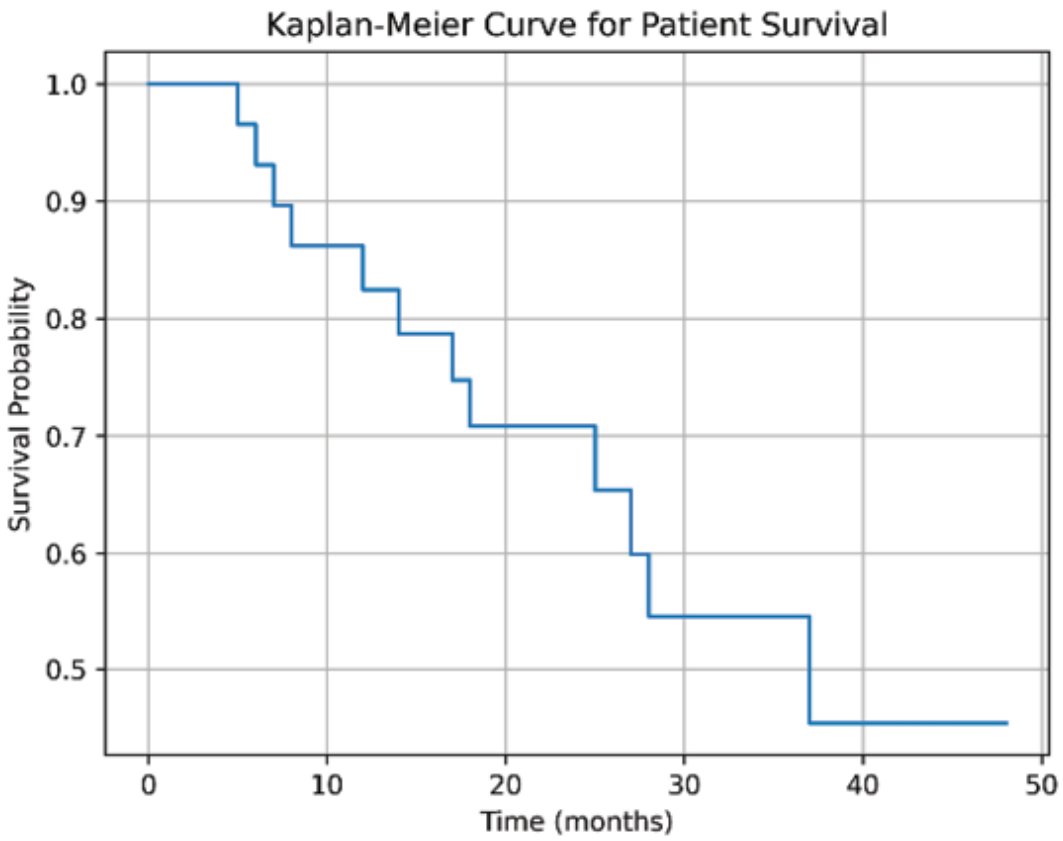

Figure 9.3: *Kaplan-Meier curve showing change in probability of survival over time*

Following is an example on survival analysis project:

Analyzes and demonstrates patient survival after surgery on a fictitious dataset of patients who have undergone a specific type of surgery. The goal is to understand the factors that affect patient survival time after surgery. Specifically, to analyze the questions. What is the overall survival rate of patients after surgery? How does survival vary with patient age? Is there a significant difference in survival between men and women?

The data includes the following columns:

Columns	Description
patient_id	Unique identifier for each patient
surgery_date	Date of the surgery

event	Indicates whether the event of interest (death) occurred (1) or not (0) during the follow-up period (censored)
survival_time	Time (in days) from surgery to the event (if it occurred) or the end of the follow-up period (if censored).

***Table 9.3**: Surgery patient dataset column details*

Tutorial 9.17: To implement Kaplan-Meier survival curve analysis of overall patient survival after surgery, is as follows:

```
import pandas as pd
from lifelines import KaplanMeierFitter
# Create sample data for 30 patients
sample_data = {
    'patient_id': list(range(101, 131)),  # Patient IDs from 101 to 130
    'surgery_date': [
        '2020-01-01', '2020-02-15', '2020-03-05', '2020-04-10', '2020-05-20',
        '2020-06-10', '2020-07-25', '2020-08-15', '2020-09-05', '2020-10-20',
        '2020-11-10', '2020-12-05', '2021-01-15', '2021-02-20', '2021-03-10',
        '2021-04-05', '2021-05-20', '2021-06-10', '2021-07-25', '2021-08-15',
        '2021-09-05', '2021-10-20', '2021-11-10', '2021-12-05', '2022-01-15',
        '2022-02-20', '2022-03-10', '2022-04-05', '2022-05-20', '2022-06-10'],
    # 1 for death and 0 for censored
    'event': [1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1,
              1, 0, 1, 0, 1, 1, 0, 1, 0, 1],
    # Survival time in days
```

```
    'survival_time': [365, 730, 180, 540, 270, 300, 600, 150, 450, 240,
                      330, 720, 210, 480, 270, 660, 150, 390, 210, 570,
                      240, 330, 720, 180, 420, 240, 600, 120, 360, 210],
    # Gender 0 (Male)  and 1 (Female)
    'gender': [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
               1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    # Age Group 0 - 40 Years is 1 and 41+  Years is 2
    'age_group': [1, 2, 1, 2, 1, 2, 2, 1, 2, 1, 1, 2, 1, 2, 1, 2, 2, 1, 2, 2,
                  1, 2, 1, 2, 1, 1, 2, 1, 2, 1]
}
# Create the dataframe
data = pd.DataFrame(sample_data)
# Initialize the Kaplan-Meier estimator
kmf = KaplanMeierFitter()
# Fit the survival data
kmf.fit(data['survival_time'], event_observed=data['event'],
        label='Overall Survival Analysis')
# Plot the survival curve
kmf.plot()
plt.xlabel("Time (days)")
plt.ylabel("Survival Probability")
plt.title("Kaplan-Meier Curve for Patient Survival")
plt.savefig('example_overall_analysis.png', dpi=600, bbox_inches='tight')
plt.show()
```

Output:

In *Figure 9.4*, survival curve (line) shows the decline in the probability of survival over time, with a steep drop from 100 to 400 days. The widening of the CI, which are the shaded area, indicate greater uncertainty in the survival estimate at those time intervals:

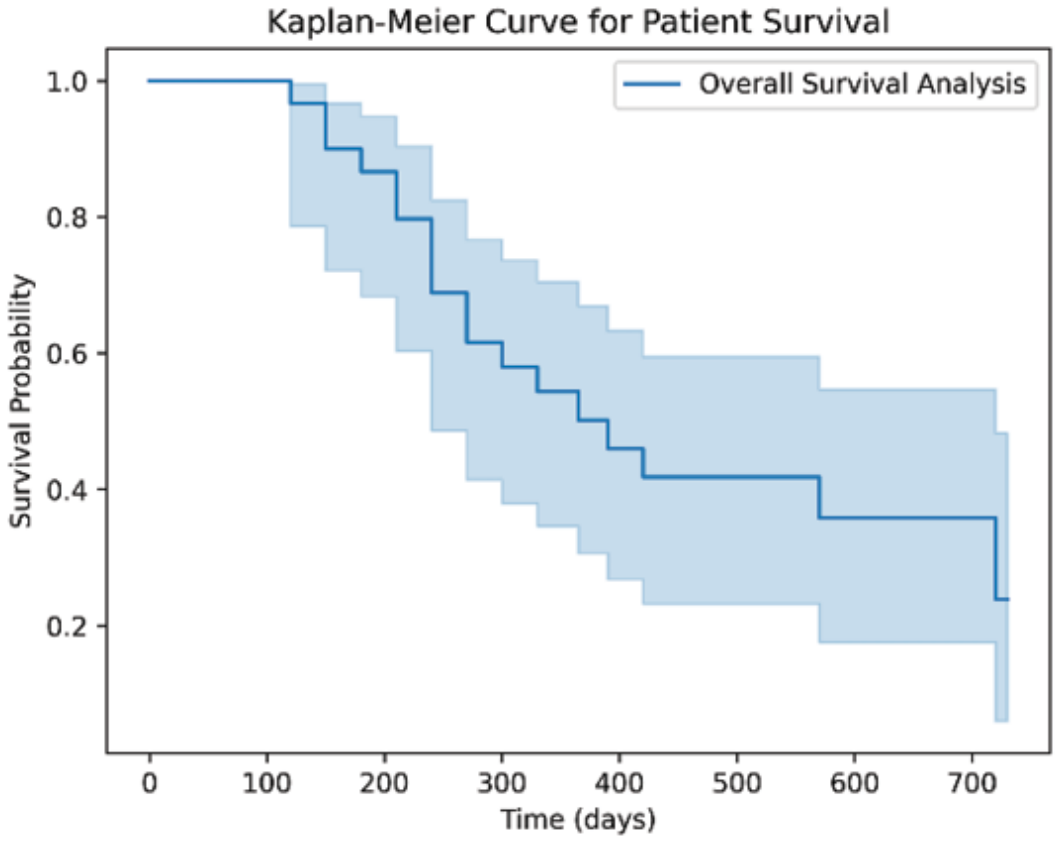

***Figure 9.4**: Kapl,an-Meier curve of overall post-surgery survival*

Tutorial 9.18: To continue *Tutorial 9.17,* estimate Kaplan-Meier survival curve analysis for the two age groups after surgery, as follows:

```
# Separate data by gender groups
age_group_1 = data[data['age_group'] == 1]
# Fit survival data for Gender 1 (Male) age group
kmf_age_1 = KaplanMeierFitter()
kmf_age_1.fit(age_group_1['survival_time'],
              event_observed=age_group_1['event'], label='A
```

```
ge Group 0 - 40 Years')
# Fit survival data for Gender 2 (Female) age group
age_group_2 = data[data['age_group'] == 2]
kmf_age_2 = KaplanMeierFitter()
kmf_age_2.fit(age_group_2['survival_time'],
              event_observed=age_group_2['event'], label='A
ge Group 41+  Years')
# Plot the survival curve for both age groups
kmf_age_1.plot()
kmf_age_2.plot()
plt.xlabel("Time (days)")
plt.ylabel("Survival Probability")
plt.title("Survival Curve by Age Groups")
plt.savefig('example_analysis_age_group.png', dpi=600,
bbox_inches='tight')
plt.show()
```

Output:

Figure 9.5 survival curve shows age group 41+ years has lower survival probability then age group 0 to 40:

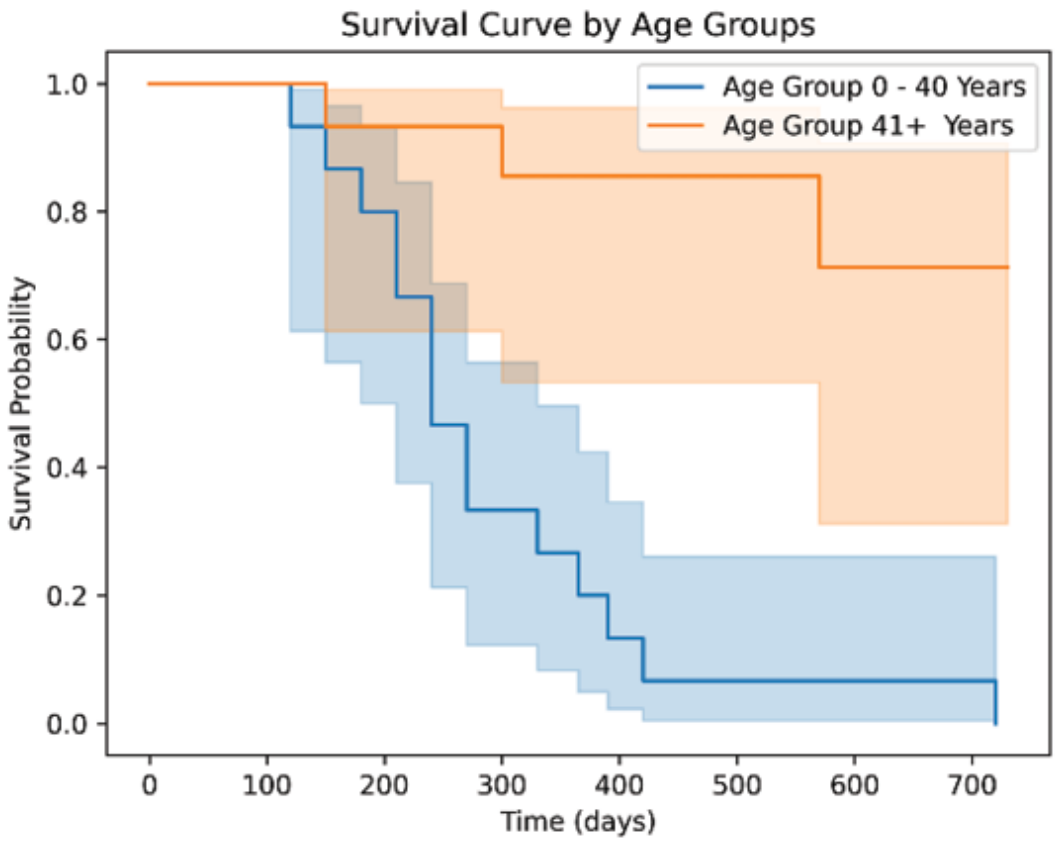

***Figure 9.5**: Kaplan-Meier curve of post-surgery survival by age group*

Tutorial 9.19: To continue *Tutorial 9.17*, estimate Kaplan-Meier survival curve analysis for the two gender groups after surgery, is as follows:

```
# Separate data by gender groups
gender_group_0 = data[data['gender'] == 0]
# Fit survival data for Gender 0 (Male) group
kmf_gender_0 = KaplanMeierFitter()
kmf_gender_0.fit(gender_group_0['survival_time'],
                 event_observed=gender_group_0['event'], label='Gender 0 (Male)')
# Fit survival data for Gender 1 (Female) group
gender_group_1 = data[data['gender'] == 1]
kmf_gender_1 = KaplanMeierFitter()
kmf_gender_1.fit(gender_group_1['survival_time'],
```

```
            event_observed=gender_group_1['event'], label='Gender 1 (Female)')
# Plot the survival curve for both age groups
kmf_gender_0.plot()
kmf_gender_1.plot()
plt.xlabel("Time (days)")
plt.ylabel("Survival Probability")
plt.title("Survival Curve by Gender")
plt.savefig('example_analysis_gender_group.png', dpi=600, bbox_inches='tight')
plt.show()
```

Output:

Figure 9.6 survival curve shows female has lower survival probability than male:

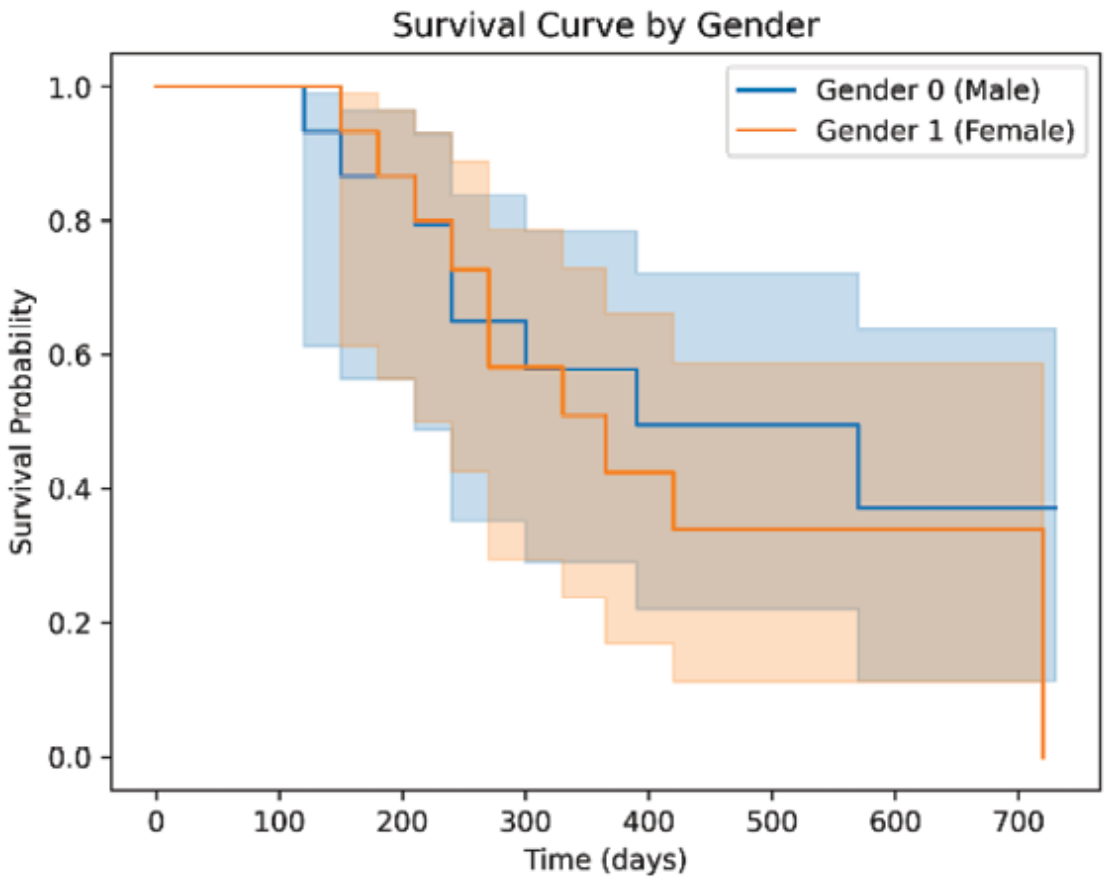

***Figure 9.6**: Kaplan-Meier curve showing of post-surgery survival by gender*

Time series analysis

Time series analysis is a powerful statistical technique used to analyze data collected over time. It helps identify patterns, trends and seasonality in data. Imagine a sequence of data points, such as daily temperatures or monthly sales figures, ordered by time. Time series analysis allows you to make sense of these sequences and potentially predict future values. The data used in time series analysis consists of measurements taken at consistent time intervals. This can be daily, hourly, monthly, or even yearly, depending on the phenomenon being studied. Then the goal is to extract meaningful information from the data. This includes the following techniques:

- **Identifying trends**: Are the values increasing, decreasing, or remaining constant over time?
- **Seasonality**: Are there predictable patterns within a specific time frame, like seasonal fluctuations in sales data?
- **Stationarity**: Does the data have a constant mean and variance over time, or is it constantly changing?

Once you understand the patterns in the data, you can use time series analysis to predict future values. This is critical for applications as diverse as predicting sales trends, stock prices, or weather patterns. For example, to analyze a store's sales data. Imagine you are a retail store manager and you have daily sales data for the past year. Time series analysis can help you do the following:

- **Identify trends**: Are your overall sales increasing or decreasing over the year? Are there significant upward or downward trends?
- **Seasonality**: Do sales show a weekly or monthly pattern? Perhaps sales are higher during holidays or

certain seasons.

- **Forecasting:** Based on the trends and seasonality you identify, you can forecast sales for upcoming periods. This can help you manage inventory, make staffing decisions, and plan marketing campaigns.

By understanding these aspects of your sales data, you can make data-driven decisions to optimize your business strategies. *Tutorial 9.17*, *Tutorial 9.18*, *Tutorial 9.19* show the time series analysis of sales data for trend analysis, seasonality, basic forecasting.

Tutorial 9.20: To implement time series analysis of sales data for trend analysis, is as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
# Sample sales data
data = pd.DataFrame({
    'date': pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04', '2023-01-05',
                            '2023-01-06', '2023-01-07', '2023-01-08', '2023-01-09', '2023-01-10',
                            '2023-02-01', '2023-02-02', '2023-02-03', '2023-02-04', '2023-02-05',
                            '2023-02-06', '2023-02-07', '2023-02-08', '2023-02-09', '2023-02-10',
                            '2023-03-01', '2023-03-02', '2023-03-03', '2023-03-04', '2023-03-05',
                            '2023-03-06', '2023-03-07', '2023-03-08', '2023-03-09', '2023-03-10',
                            '2023-04-01', '2023-04-02', '2023-04-03', '2023-04-04', '2023-04-05',
                            '2023-04-06', '2023-04-07', '2023-04-08', '2023-04-09', '2023-04-10'
                            ]),
```

```
    'sales': [100, 80, 95, 110, 120, 90, 130, 100, 115, 125,
              140, 130, 110, 100, 120, 95, 145, 110, 105, 130,
              150, 120, 110, 100, 135, 85, 150, 100, 120, 140,
              160, 150, 120, 110, 100, 130, 105, 140, 125, 150]
})
# Set the 'date' column as the index
data.set_index('date', inplace=True)
# Plot the time series data
data['sales'].plot(figsize=(12, 6))
plt.xlabel('Date')
plt.ylabel('Sales')
plt.title('Sales Over Time')
plt.savefig('trendanalysis.png', dpi=600, bbox_inches='tight')
plt.show()
```

Output:

Figure 9.7 shows overall sales increasing over the year, with upward trends:

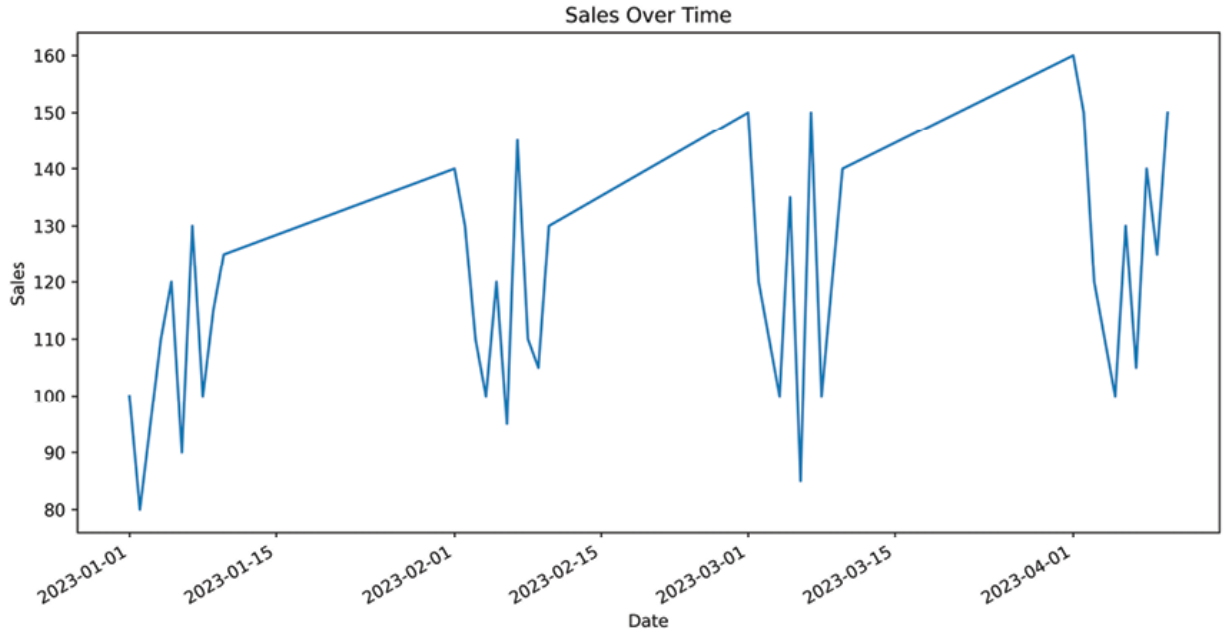

***Figure 9.7**: Time series analysis to view sales trends throughout the year*

Tutorial 9.21: To implement time series analysis of sales data over season or month, to see if season, holidays or festivals affect sales, is as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
# Sample sales data
data = pd.DataFrame({
    'date': pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04', '2023-01-05',
                            '2023-01-06', '2023-01-07', '2023-01-08', '2023-01-09', '2023-01-10',
                            '2023-02-01', '2023-02-02', '2023-02-03', '2023-02-04', '2023-02-05',
                            '2023-02-06', '2023-02-07', '2023-02-08', '2023-02-09', '2023-02-10',
                            '2023-03-01', '2023-03-02', '2023-03-03', '2023-03-04', '2023-03-05',
                            '2023-03-06', '2023-03-07', '2023-03-08', '2023-03-09', '2023-03-10',
```

```
                    '2023-04-01', '2023-04-02', '2023-04-03', '2023-04-04', '2023-04-05',
                    '2023-04-06', '2023-04-07', '2023-04-08', '2023-04-09', '2023-04-10'
                    ]),
    'sales': [100, 80, 95, 110, 120, 90, 130, 100, 115, 125,
              140, 130, 110, 100, 120, 95, 145, 110, 105, 130,
              150, 120, 110, 100, 135, 85, 150, 100, 120, 140,
              160, 150, 120, 110, 100, 130, 105, 140, 125, 150]
})
# Set the 'date' column as the index
data.set_index('date', inplace=True)
# Resample data by month (or other relevant period) and calculate mean sales
monthly_sales = data.resample('M')['sales'].mean()
monthly_sales.plot(figsize=(10, 6))
plt.xlabel('Month')
plt.ylabel('Average Sales')
plt.title('Monthly Average Sales')
plt.savefig('seasonality.png', dpi=600, bbox_inches='tight')
plt.show()
```

Output:

Figure 9.8 shows overall sales increasing over the year with upward trends:

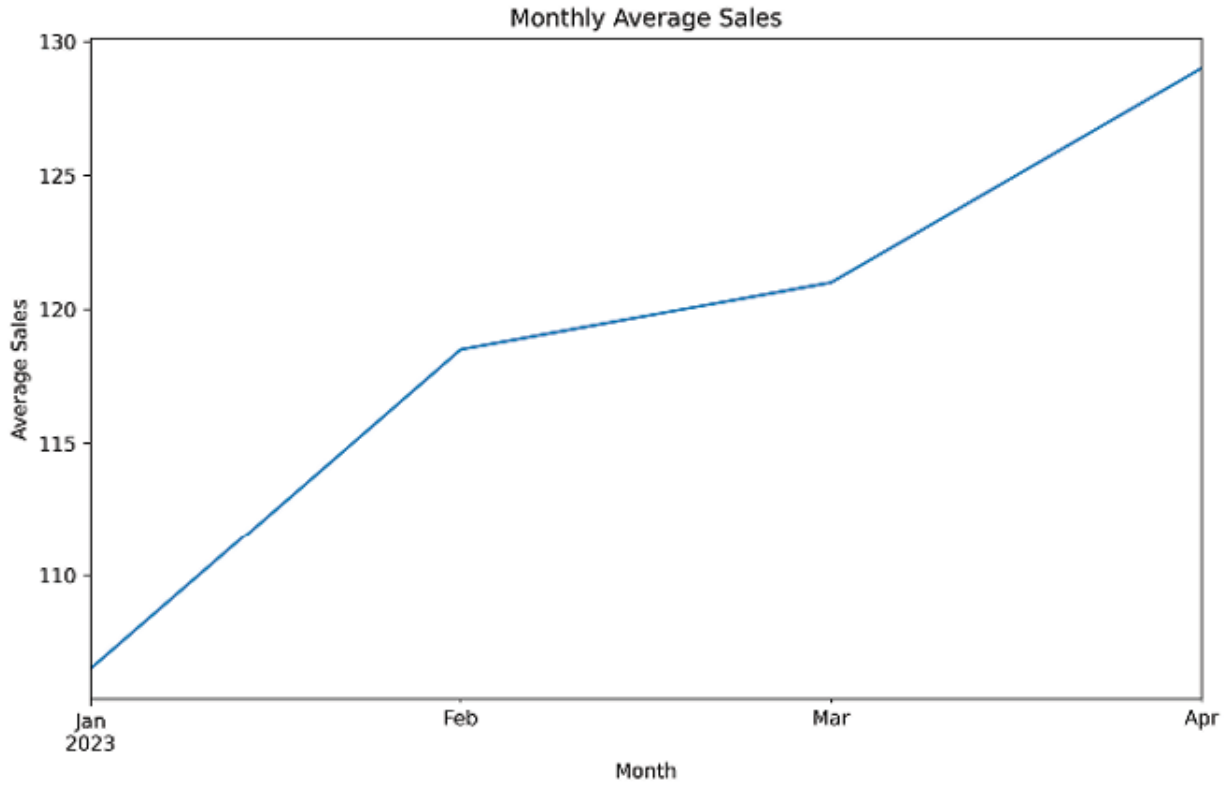

Figure 9.8: *Time series analysis of sales by month*

Tutorial 9.22: To implement time series analysis of sales data for basic forecasting, is as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
# Sample sales data
data = pd.DataFrame({
    'date': pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04', '2023-01-05',
                            '2023-01-06', '2023-01-07', '2023-01-08', '2023-01-09', '2023-01-10',
                            '2023-02-01', '2023-02-02', '2023-02-03', '2023-02-04', '2023-02-05',
                            '2023-02-06', '2023-02-07', '2023-02-08', '2023-02-09', '2023-02-10',
                            '2023-03-01', '2023-03-02', '2023-03-
```

```
03', '2023-03-04', '2023-03-05',
                   '2023-03-06', '2023-03-07', '2023-03-08', '2023-03-09', '2023-03-10',
                   '2023-04-01', '2023-04-02', '2023-04-03', '2023-04-04', '2023-04-05',
                   '2023-04-06', '2023-04-07', '2023-04-08', '2023-04-09', '2023-04-10'
                   ]),
    'sales': [100, 80, 95, 110, 120, 90, 130, 100, 115, 125,
              140, 130, 110, 100, 120, 95, 145, 110, 105, 130,
              150, 120, 110, 100, 135, 85, 150, 100, 120, 140,
              160, 150, 120, 110, 100, 130, 105, 140, 125, 150]
})
# Set the 'date' column as the index
data.set_index('date', inplace=True)
# Calculate a simple moving average with a window of 7 days
data['rolling_avg_7'] = data['sales'].rolling(window=7).mean()
data[['sales', 'rolling_avg_7']].plot(figsize=(12, 6))
plt.xlabel('Date')
plt.ylabel('Sales')
plt.title('Sales with 7-Day Moving Average')
plt.savefig('basicforecasting.png', dpi=600, bbox_inches='tight')
plt.show()
```

Output:

In Figure 9.9, the solid gray represents the daily sales data. The dashed dark gray line represents the rolling average of sales over a seven-day window. The dashed dark gray line (rolling average) above the solid gray sales line indicates an upward trend in sales over the seven-day period. This indicates that recent sales are higher than the average. The opposite is a downward trend. As you can see, changes in the slope of the rolling average (i.e. sudden spikes or declines)

reveal shifts in sales patterns.

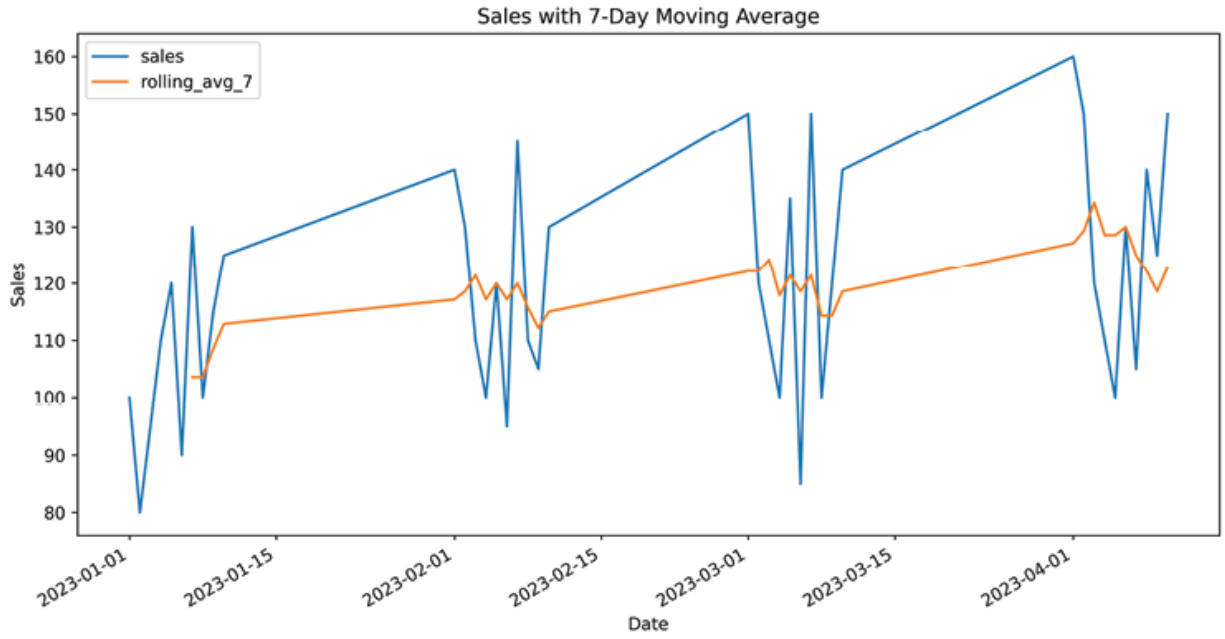

***Figure 9.9**: Time series analysis of monthly sales to assess the impact of seasons, holidays, and festivals*

Conclusion

Finally, this chapter served as an engaging exploration of powerful data analysis techniques like linear algebra, nonparametric statistics, time series analysis and survival analysis. We experienced the elegance of linear algebra, the foundation for maneuvering complex data structures. We embraced the liberating power of nonparametric statistics, which allows us to analyze data without stringent assumptions. We ventured into the realm of time series analysis, revealing the hidden patterns in sequential data. Finally, we delved into survival analysis, a meticulous technique for understanding the time frames associated with the occurrence of events. This chapter, however, serves only as a stepping stone, providing you with the basic knowledge to embark on a deeper exploration. The path to data mastery requires ongoing learning and

experimentation.

Following are some suggested next steps to keep you moving forward: deepen your understanding through practice by tackling real-world problems, master software, packages, and tools and embrace learning. *Chapter 10, Generative AI and Prompt Engineering* ventures into the cutting-edge realm of GPT-4, exploring the exciting potential of prompt engineering for statistics and data science. We will look at how this revolutionary language model can be used to streamline data analysis workflows and unlock new insights from your data.